PVS Theorem Proving Enhancements

Final Report
For the Period September 23 1996 through March 24, 1997
Contract No. N00014-96-C-2106
SRI Project ECU 1513

Prepared by:
Natarajan Shankar,
Senior Computer Scientist
Computer Science Laboratory

Prepared for:
Naval Research Laboratory
Code 5546
4555 Overlook Ave. SW
Washington, DC 20375-5000
Attn: Ms Myla Archer, COR

Approved:
Patrick D. Lincoln, Director
Computer Science Laboratory

Donald L. Nielson, Vice President
Computing and Engineering Sciences Division

Approved for public release
Distribution Unlimited
Summary: The goal of the project was to augment PVS with features that simplified the construction and management of proofs, and to document the PVS functions needed for writing proof strategies. The extensions to PVS developed in this project include

(Task 2.1) Multiple-proof maintenance: Each formula can now retain multiple proofs. Each proof is assigned a name. PVS now provides Emacs support for selecting, browsing, editing, and rerunning proofs. This is described in Section 1.

(Task 2.2) Comments in proofs: A \texttt{COMMENT} command associates a comment with a proof sequent. Comments can be supplied by the user or generated within strategies. This is described in Section 2.

(Task 2.3) Labeling and accessing sequent formulas: The newly added \texttt{LABEL} command can be used to label selected sequent formulas for future access. The \texttt{WITH-LABELS} command applies a rule and labels the new formulas in the resulting subgoals from a list of labels, where each list of labels applies to the new formulas in one subgoal. This is described in Section 3.

The primitives for selecting sequent formulas or their numbers based on selection predicates have also been documented. This documentation appears in Section 4.

(Task 2.4) Rerunning proofs with checkpoints: A selected proof can be edited to insert checkpoints, and rerun so that the uncheckpointed parts of the proof are not rerun, and the user is prompted at the checkpoints. Checkpointing is described in Section 5.

(Task 2.5) Deconstructing \texttt{EXPAND}: The \texttt{EXPAND} command has been augmented so that it can be directed to not automatically simplify the definition expansion. The updated documentation for the \texttt{EXPAND} rule appears in Section 6.

(Task 2.6) Saved \texttt{SKIP} command: The \texttt{APPLY} command takes a \texttt{SAVE?} option that can be used to retain the applied step even if it is unsuccessful. This can be used to set the values of global variables for use later in the proof. It also takes a \texttt{TIME?} flag, which can be used to obtain timing information about the proof steps being applied. These enhancements are described in Section 7.

1 Multiple Proofs

PVS now supports multiple proofs for a given formula. When a proof attempt is completed, either by quitting or successfully completing the proof, the proof is checked for changes. If any changes have occurred, the user is queried about whether to save the proof, and whether to overwrite the current proof or to create a new proof. If a new proof is created, the user is prompted for a proof identifier and description.

In addition to a proof identifier, description, and proof script, the proof objects contain the status, the date of creation, the date last run, and the run time.

Every formula that has proofs has a default proof, which is used for most of the existing commands, such as prove, prove-theory, and status-proofchain. Whenever a proof is saved, it automatically becomes the default.
Three new Emacs commands allow for browsing and manipulating multiple proofs: display-proofs-formula, display-proofs-theory, and display-proofs-pvs-file. These commands all pop up buffers with a table of proofs. The default proof is marked with a ‘+’. Within such buffers, the following keys have the following effects.

<table>
<thead>
<tr>
<th>Key</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Change description: add or change the description for the proof</td>
</tr>
<tr>
<td>d</td>
<td>Default proof: set the default to the specified proof</td>
</tr>
<tr>
<td>e</td>
<td>Edit proof: bring up a Proof buffer for the specified proof; the proof may then be applied to other formulas</td>
</tr>
<tr>
<td>p</td>
<td>Prove: rerun the specified proof (makes it the default)</td>
</tr>
<tr>
<td>q</td>
<td>Quit: exit the Proof buffer</td>
</tr>
<tr>
<td>r</td>
<td>Rename proof: rename the specified proof</td>
</tr>
<tr>
<td>s</td>
<td>Show proof: Show the specified proof in a Proof: (id) buffer</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete proof: delete the specified proof from the formula</td>
</tr>
</tbody>
</table>

2 Comments in Proofs

The following is the excerpt from the draft PVS Prover Reference Manual:

**syntax:** (comment string)

**effect:** Attaches a comment string to the current proof sequent that is printed with preceding semicolons above the sequent formulas. This comment string is also saved with the proof. The comment command can be nested within strategies, and the comments are retained on the subgoals generated by the strategy.

**usage:** (comment "3rd induction case") : Prints the comment string ; ; ; 3rd induction case between the sequent label and the sequent formulas.

3 Labeled Sequent Formulas

Sequent formulas are labeled using the label proof command. The following is the relevant PVS documentation for this command.

**syntax:** (label string-or-symbol fnums)

**effect:** It is often useful to group and label a collection of related formulas in a proof sequent. The label command is used for this purpose. Each sequent formula can bear at most one label. The label is printed alongside the fnum whenever a proof sequent is displayed. A label can be used wherever an fnum is expected. A label can supplied as either a string, e.g., "label" or a symbol, e.g., |label|, though it is stored internally as a symbol. Labels are automatically inherited by any subformulas of a sequent formula that appear through the application of an inference rule, e.g., flatten applied
to a consequent formula $A \lor B$ labeled main results in two sequent formulas $A$ and $B$, both labeled main.

**usage:** (label "uniqueness" -3) : Labels the formula numbered -3 by the label uniqueness.

(labell "type-constraints" (-1 -3 -4)) : Labels the formulas numbered -1, -3, and -4 by the label type-constraints.

(label "antecedents" -) : Labels all the antecedent formulas with the label antecedents.

(bddsimp "type-constraints") : Applies BDD-based propositional simplification to the formulas labeled type-constraints.

**notes:** Note that the bddsimp command does not retain labels since there is no simple way to retain the connection between the formula returned by BDD-simplification and its original parent formula.

A common way to introduce labels is to immediately label the new sequent formulas generated by a proof step. The with-labels command applies a proof step and then labels the newly generated formulas. The extract from the PVS documentation is given as follows.

**syntax:** (with-labels rule labels)

**effect:** Given a proof step rule and a list of list of labels ($\langle l_{11} \ldots \rangle \ldots \langle l_{n1} \ldots \rangle$), if the rule generates $n$ subgoals, then the $j$th new sequent formula in the $i$th subgoal is assigned the label $l_{ij}$. If there are more subgoals than label lists, then the last label list is applied to the remaining subgoals. In each pairing of new formulas with labels in a list, if there are more formulas than labels, the last label is applied to the remaining new formulas. A singleton list of labels can be replaced by a single label.

**usage:** (with-labels (flatten) ("11" "12" "13"): Applies the flatten rule to the current proof subgoal and labels the new sequent formulas thus produced as 11, 12, and 13, respectively.

(with-labels (prop) ("111" "112" "113") ("121" "122"): Applies the prop rule and labels the new formulas in the first subgoal by labels 111, 112, and 113, and the new formulas in any remaining subgoals by labels 121 and 122.

(with-labels (prop) "prop-formulas"): Labels all the new sequent formulas resulting from the application of prop by the label prop-formulas.

4 Selecting Sequent Formulas

We now document the various operations on PVS data structures for terms, formulas, and proof goals that are needed for writing nontrivial PVS proof strategies. PVS data structures
are defined as classes in the Common Lisp Object System (CLOS). Each class is defined by indicating its slots. Classes can be defined as subclasses of one or more superclasses by introducing the additional slots. For example, the proof state that is the root node of a proof is defined as a subclass of an ordinary proof state that contains an extra slot for referring to the formula declaration corresponding to the proof. Data objects corresponding to a class are called instances. If a Lisp term \( t \) has instance \( v \) as its value, then \((\text{show } t)\) displays the slot values of \( v \). With PVS data structures, if value \( v \) is an instance of class \( c \), then \( c? \) is the recognizer corresponding to the class so that \((c? \ v)\) is \( T \). Furthermore, if \( c \) is a subclass of class \( b \), then \((b? \ v)\) is also \( T \). If \( s \) is a slot name in class \( c \), then \((s \ v)\) returns the corresponding slot value in \( v \). A slot value is destructively updated by \((\text{setf } (s \ v) \ u)\), which sets the slot value of slot \( s \) in \( v \) to \( u \). An instance can be nondestructively copied and updated by \((\text{copy } v \ 's1 \ u1 \ 's2 \ u2)\), which returns a copy of \( v \) with slot \( s1 \) set to \( u1 \) and \( s2 \) set to \( u2 \). There is a lazy form of copy where \((\text{lcopy } v \ 's1 \ u1 \ 's2 \ u2)\) creates a new copy only when the updates actually change the slot values.

The global variable \(*ps*\) is always bound to the currently active proof goal. Each proof goal is an instance of class \proofstate. The sequent corresponding to the proof goal is saved in the \current-goal slot so that \((\current-goal \ *ps*)\) contains the current sequent which is an instance of the class \sequent. The class \sequent contains the slots \s-forms which is the list of visible sequent formulas, and \hidden-s-forms which is the list of hidden sequent formulas. Each sequent formula is an instance of class \s-formula where the sequent formula itself is contained in the slot \formula. So, for example, \((\text{formula } (\text{car } (\s-forms (\current-goal \ *ps*))))\) returns the expression corresponding to the first sequent formula. The sequent formulas are maintained in a list. The antecedent formulas appear negated in this list.

Several Lisp functions select sequent formulas given their labels or numbers, or collect the numbers of selected sequent formulas. Given a sequent \seq, typically obtained by \((\s-forms (\current-goal \ *ps*))\) and a list of labels or formula numbers \fnums, the Lisp expression \((\text{select-seq } \seq \ \fnums)\) returns the list of sequent formulas in \seq corresponding to the given \fnums. The Lisp expression \((\text{delete-seq } \seq \ \fnums)\) returns the list of sequent formulas in \seq that are not selected by the given \fnum. If we are interested in selecting the sequent formulas according to some predicate, then the Lisp expression \((\text{gather-seq } \seq \ \text{yes-fnums no-fnums pred})\) returns the list of sequent formulas in \seq that are selected by \yes-fnums but not by \no-fnums such that the formula part of the sequent formula satisfies the unary predicate given by \pred. Note that the formula numbers given by \fnums can also be \'\* (for all the formulas), \'+' (for the consequent formulas), and \'\-' (for the antecedent formulas), and also formula labels.

Since many commands take formula numbers or lists of formula numbers as arguments, it is useful to select these numbers rather than the formulas themselves. The Lisp expression \((\text{gather-fnums } \seq \ \text{yes-fnums no-fnums pred})\) returns the list of all the formula numbers of sequent formulas in \seq corresponding to \fnums that satisfy the predicate \pred on the formula part of a sequent formula.

Typical formulas are either negations, disjunctions, conjunctions, implications, equalities, equivalences, conditional expressions, arithmetic inequalities, or universally or existentially quantified expressions. Quantified expressions are in the class \binding-expr with slots bindings which returns the bound variables, and \expression, which returns the body.
of the binding expression. The other forms are all instances of the application class consisting of a slot for the operator and one for the argument. The first or only argument of an application expr can be obtained by (args1 expr). The second argument, if any, can be obtained by (args2 expr). The predicates for recognizing the different connectives are summarized in the following table.

<table>
<thead>
<tr>
<th>Connective</th>
<th>Recognizer Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>(negation? expr)</td>
</tr>
<tr>
<td>Disjunction</td>
<td>(disjunction? expr)</td>
</tr>
<tr>
<td>Conjunction</td>
<td>(conjunction? expr)</td>
</tr>
<tr>
<td>Implication</td>
<td>(implication? expr)</td>
</tr>
<tr>
<td>Equality</td>
<td>(equality? expr)</td>
</tr>
<tr>
<td>Equivalence/Equality</td>
<td>(iff? expr)</td>
</tr>
<tr>
<td>Conditional</td>
<td>(branch? expr)</td>
</tr>
<tr>
<td>Universal Formula</td>
<td>(forall-expr? expr)</td>
</tr>
<tr>
<td>Existential Formula</td>
<td>(exists-expr? expr)</td>
</tr>
</tbody>
</table>

Thus, the Lisp expression

(gather-seq (s-forms (current-goal *ps*))
  
    nil
    #'(lambda (expr) (and (negation? expr)
                      (forall-expr? (args1 expr)))))

collects the list of universally quantified antecedent formulas, and the Lisp expression

(gather-fnums (s-forms (current-goal *ps*))
  
    nil
    #'(lambda (expr) (and (negation? expr)
                       (forall-expr? (args1 expr)))))

returns the corresponding list of formula numbers.

5 Checkpointing Proofs

Checkpoints may be added to the Proof buffer obtained by the edit-proof command. To add a checkpoint, position the cursor and type C-c a. The checkpoint is indicated by a double exclamation point (!!). Any number of checkpoints may be added. When the proof is installed using C-c C-i, these are changed to the checkpoint proof rule, and branches of the proof that do not have a checkpoint on them are wrapped in a just-install-proof proof rule. When this proof is rerun, it will run until it hits a checkpoint, and then prompt for a prover command. When it hits a just-install-proof, it simply installs the given commands and marks that branch as proved. This allows the prover to quickly get to the
next checkpoint, without attempting to reprove branches that do not have checkpoints in them. When a proof that has just-install-proof rules in it is finished, the prover asks whether the proof should be rerun, as the formula will not be considered proved until the proof is rerun.

To remove a checkpoint from the Proof buffer, position the cursor at the checkpoint and type C-c r. To remove all checkpoints, type C-c DEL.

6 Enhanced EXPAND Rule

The EXPAND command has been augmented so that it can be directed to not apply any simplification to the formulas resulting from definition expansion. The revised PVS documentation is as follows.

**syntax:** (expand name &optional fnum[*] occurrence if-simplifies assert?)

**effect:** Expands (and simplifies) the definition of name at a given occurrence. If occurrence is not given, then all instances of the definition are expanded. The occurrence is given as a number n referring to the nth occurrence of the function symbol counting from the left, or as a list of such numbers. If the if-simplifies flag is t, then any expansion within a sequent formula occurs only if the expanded form can be simplified (using the decision procedures). The if-simplifies flag is needed to control infinite expansions in case expand is used repeatedly inside a strategy. In the default case when assert? is NIL, expand applies the simplify step with the default settings to any sequent formula in which a definition is expanded. When assert? is T, expand applies the assert version of simplify to any sequent formulas affected by definition expansion. This latter option must be exercised for compatibility with PVS 1.x. In PVS 2.1, there is a new option where the assert? flag can be NONE in which case no simplification is applied to the sequent formula following expansion.

**usage:** (expand "sum") : Expands the definition of sum throughout the current sequent, whether it simplifies or not. The resulting expressions are all simplified using decision procedures and rewriting.

(expand "sum" 1) : Expands sum throughout the formula labeled 1.

(expand "sum" 1 2) : Expands the second occurrence of sum in the formula labeled 1.

(expand "sum" :if-simplifies t) : Expands those occurrences of sum whose definitions can be simplified by means of the decision procedures. This is relevant only in the situation where the definition is a CASES or IF expression. The definition expansion occurs only if such an expression simplifies to one of its branches.

(expand "sum" :assert? T) : Expands sum, but uses assert instead of simplify in the simplification process.
errors: Occurrence ... must be nil, a positive number or a list of positive numbers: Self-explanatory.

notes: Typically, the defined rule rewrite can be used instead of expand but expand has some advantages:

- expand is faster, since definitions are simple (unconditional) equations.
- expand does not require name to be fully resolved; it can use the occurrence to get the type information needed.
- expand allows a specific occurrence or occurrences of a function symbol to be expanded.
- expand can rewrite subterms containing variables that are bound in some superterm — for example, if \( f(x) \) is defined as \( g(h(x)) \), then expand would be able to rewrite \( (\forall x.f(x) = 0) \) as \( (\forall x.g(h(x)) = 0) \), but rewrite would not.

7 Enhanced APPLY Command

The APPLY command has been enhanced with two new options for saving the command and for recording the time taken by the command. The revised PVS documentation is as follows.

**syntax:** (apply strategy &optional comment save? time?)

**effect:** The apply rule takes an application of a proof strategy and applies it as a single atomic step that generates those subgoals left unproved by the proof strategy. The apply rule is frequently used when one wishes to employ a proof strategy but is not interested in the details of the intermediate steps. A number of defined rules employ apply to suppress trivial details. The optional comment field can be used to provide a format string to be used as commentary while printing out the proof. If the save? flag is set to T, the apply step is saved even if the applied strategy results in no change to the proof. This is useful if, for example, the command within the apply uses the lisp command to change a Lisp variable for use elsewhere in the proof. The time? flag when set to T causes the apply command to return timing information regarding the applied step.

**usage:** (apply (then* (skolem 2 ("a4" "b5")) (beta) (flatten)
"Skolemizing and beta-reducing")
"The then* strategy performs each of the steps given by its arguments in sequence. Wrapping this strategy in an apply ensures that the intermediate steps in the sequence are hidden. The given commentary string is printed out as part of the proof.

(apply (try (skolem!) (flatten) (ground))) : This applies a strategy that applies (skolem!) to the current goal, and if that succeeds applies (flatten) to the resulting subgoals; otherwise, it applies (ground) to the current goal. The rule carries out this strategy in an atomic step and returns the resulting subgoals.
(apply (grind) :save? T :time? T) This applies the grind strategy but saves the step even when grind has no effect, and returns timing information.

errors: No error messages are generated.

8 Conclusion

The support for multiple proofs makes it easier to experiment with different proof styles without losing earlier proof attempts. Proof comments and labels provide both human and system robustness in the development of specifications and their proofs. Allowing proofs to be rerun using checkpoints can significantly speed up proof development, and the enhancements to the SKIP command allow the user more control over proofs, as it may be used to set global variables that are subsequently used in other strategies. The new facilities described above are a significant enhancement to PVS.